Look It Up: Real-Life Database Indexing

Christophe Pettus
PostgreSQL Experts
PgConf.NYC 2023
Christophe Pettus
CEO, PGX Inc.

christophe.pettus@pgexperts.com

twitter @xof
Indexes!

• We don’t need indexes.

• By definition!

• An index never, ever changes the actual result that comes back from a query.

• A 100% SQL Standard-compliant database can have no index functionality at all.

• So, why bother?
O(N)
O(N)

- Without indexes, all queries are sequential scans (at best).
- This is horrible, terrible, bad, no good.
- The point of an index is to turn O(N) into O(something better than N).
  - Ideally O(logN) or O(1)
- But…
Just a reminder.

• Indexes are essential for database performance, but…

• … they do not result in speed improvements in all cases.

• It’s important to match indexes to the particular queries, datatypes, and workloads they are going to support.

• That being said…

• … let’s look at PostgreSQL’s amazing indexes!
The Toolbox.

- B-Tree.
- Hash.
- GiST.
- GIN.
- SP-GiST.
- BRIN.
- Bloom.
Wow.

- PostgreSQL has a wide and amazing range of index types.
- Each has a range of queries and datatypes that they work well for.
- But how do you know which one to use?
- Someone should give a talk on that.
B-Tree.
B-Tree Indexes.

- The most powerful algorithm in computer science whose name is a mystery.
- Balanced? Broad? Boeing? Bushy? The one that came after A-Tree indexes?
- Old enough to be your parent: First paper published in 1972.
- The “default” index type in PostgreSQL (and pretty much every other database, everywhere).
It’s that graphic again.

By CyHawk - Own work based on https://dl.acm.org/citation.cfm?doid=356770.356776.
CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=11701365
So many good things.

- B-Trees tend to be very shallow compared to other tree structures.
  - Shallow structures mean fewer disk page accesses.
- Provide $O(\log N)$ access to leaf notes.
- Easy to walk in ordered directions, so can help with ORDER BY, merge joins…
B-Trees, PostgreSQL Style.

• PostgreSQL B-Trees have a variable number of keys per node…

  • … since PostgreSQL has a wide range of indexable types.

• Entire key value is copied into the index.

• Larger values means fewer keys per node, so deeper indexes.
Recent Improvements

- Significant improvements to B-Tree structure.
- Smaller indexes, especially with many duplicate keys.
- Requires that the index be reconstructed if it exists already.
  - A quick REINDEX CONCURRENTLY will handle it.
Perfect! We’re Done.

• Not so fast.

• “Entire key value is copied into the index.”

• Not good (or not available) for long data types.

• Requires a totally-ordered type (one that supports =, <, > for all values).

• Many, many datatypes are not totally-ordered.
Hash.
Hash Indexes.

• Converts the input value to a 32-bit hash code.

• Hash table points to buckets of row pointers.

• Works on data of arbitrary length.
Making a hash of it.

- Only supports one operator: =.
  - But that’s a pretty important operator.
- Indexes are smaller than B-Tree, especially for large key values.
  - Access can be faster, too, if there are few collisions.
- Great for long values on which equality is the primary operation.
  - URLs, long hash values (from other algorithms), etc.
GiST.
GiST Indexes.

- GiST is a framework, not a specific index type.
- GiST is a generalized framework to make it easy to write indexes for any data type.
- What a GiST-based index does depends on the particular type being indexed.
- For example:
Generalized Search Tree.

- Can be used for any type where “containment” or “proximity” is a meaningful operation.

- Standard total ordering can be considered a special case of proximity[^citation required].

- Ranges, geometric types, text trigrams, etc., etc…

- Not as efficient as B-Tree for classic scalar types with ordering, or for simple equality comparisons.
GIN.
General Inverted Index.

• Both B-Tree and GiST perform poorly where there are lots and lots of identical keys.

• However, full text search (as the most classic case) has exactly that situation.

• A (relatively) small corpus of words with a (relatively) large number of records and positions that contain them.

• Thus, GIN!
A Forest of Trees.

- GIN indexes organize the keys (e.g., normalized words) into a B-Tree.

- The “leaves” of the B-Tree are lists or B-Trees themselves of pointers to rows that hold them.

- Scales very efficiently for a large number of identical keys.

  - Full-text search, indexing array members and JSON keys, etc.
SP-GiST.
Space Partitioning GiST.

- Similar to GiST in concept: A framework for building indexes.
- Has a different range of algorithms for partitioning than “classic” GiST.
- Designed for situations where a classic GiST index would be highly unbalanced.
- More later!
BRIN.
Block-Range Index.

• B-Tree indexes can be very large.
  
  • Not uncommon for the indexes in a database to exceed the size of the heap.
  
• B-Trees assume we know nothing about a correlation between the index key and the location of the row in the table.
  
• But often, we do know!
created_at timestamp
default now()

• Tables that are INSERT-heavy often have monotonically increasing keys (SERIAL primary keys, timestamps)...

• ... and if the tables are not UPDATE-heavy, the key will be strongly correlated with the position of the row in the table.

• BRIN takes advantage of that.
BRIN it on.

• Instead of a tree of keys, records ranges of keys and pages that (probably) contain them.

• Much, much smaller than a B-Tree index.

• If the correlation assumption is true, can be much faster to retrieve ranges (like, “get me all orders from last year”) than a B-Tree.

• Not good for heavily-updated tables, small tables, or tables without a monotonically-increasing index key.
Bloom.
Bloom Filters

• Like a hash, only different!

• Most useful for indexing multiple columns at once.

• Very fast for multi-column searches.
  • Multiple attributes, each expressed as its own column.

• A small fraction of the size of multiple B-Tree indexes.
  • Potentially faster for a large number of attributes.
Pragmatic Concerns
Do you need an index at all?

- Indexes are expensive.
  - Slow down updates, increase disk footprint size, slow down backups / restores.
- As a very rough rule of thumb, an index will only help if less than 15-20% of the table will be returned in a query.
- This is the usual reason that the planner isn’t using a query.
Good Statistics.

- Good planner statistics are essential for proper index usage.

- Make sure tables are getting ANALYZEd and VACUUMEd.

- Consider increasing the statistics target for specific columns that have:
  
  - A lot of distinct values.
  
  - More distribution than 100 buckets can capture (UUIDs, hex hash values, tail-entropy text strings).

- Don’t just slam up statistics across the whole database!
Bad Statistics.

- 100,000,000 rows, 100 buckets, field is not UNIQUE, 25,000 distinct values.

- SELECT * FROM t WHERE sensor_id='38aa9f2c-3e5d-4dfe-9ed7-e136b567e4e2'

- Planner thinks 1m rows will come back, and may decide an index isn’t useful here.

- Setting statistics higher will likely generate much better plans.
Indexes and MVCC.

• Indexes store every version of a tuple until VACUUM cleans up dead ones.

• The HOT optimization helps, but does not completely eliminate this.

• This means that (in the default case) index scans have to go out to the heap to determine if a tuple is visible to the current transaction.

• This can significantly slow down index scans.
Index-Only Scans.

• If we know that every tuple on a page is visible to the current transaction, we can skip going to the heap.

• PostgreSQL uses the visibility map to determine this.

• If the planner thinks “enough” pages are completely visible, it will plan an Index-Only Scan.

• Nothing you have to do; the planner handles this.

  • Except: Make sure your database is getting VACUUMed properly!
Lossy Index Scans.

• Some index scans are “lossy”: It knows that some tuple in the page it is getting probably matches the query condition, but it’s not sure.

• This means that it has to retrieve pages and scan them again, throwing away rows that don’t match.

• Bitmap Index Scan / Bitmap Heap Scan are the most common type of this…

• … although some index types are inherently lossy.
Covering Indexes.

• Queries often return columns that aren’t in the indexed predicates of the query.

• Traditionally, PostgreSQL had to fetch the tuple from the heap to get those values (after all, they aren’t in the index!).

• Non-indexed columns can be added to the index… retrieved directly when the index is scanned.

  • Doesn’t help on non-Index Only Scans, and remember: you are increasing the index size with each column you add.
GIN Posting.

- GIN indexes are very fast to query, but much slower to update than other types of index.

- PostgreSQL records changes in a separate posting area, and updates the index at VACUUM time (or on demand).

- This can result in a surprising spike of activity on heavily-updated GIN indexes.

- Consider having a separate background process that calls `gin_clean_pending_list()`.
UNIQUE indexes.

• B-Trees support unique indexes.

• Optimistic insertion with recovery on index conflicts is a perfectly fine application development strategy.
  
  • ON CONFLICT … makes this much easier.

• This can be a concurrency-killer, so don’t expect very high insertion rates in the face of conflicts.

• Exclusion constraints provide a generalization of UNIQUE (“only one value that passes this comparison is allowed in this table”).
Is this a decision tree?
Is this a decision tree?
Is this a decision tree?

Yes.
What index?

• How do we decide what index to use in a particular situation?

• First, gather some information:

  • Typical queries on the table.

  • The columns, data types, and operators that are being queried.

    • Including those in JOINs.

  • How many rows the queries typically return.
How many rows?

- Does the query typically return a large percentage of the table?

  - Including “hidden” row fetches, such as COUNT(*).

- If so… an index probably won’t help!

- Refactor the query, consider summary tables or other techniques before just throwing an index at the problem.

- Small tables that fit in memory usually don’t need indexes at all, except to enforce constraints.
Which column?

- In a multi-predicate query, which column?
- Always start with the most selective predicate.
  - That is, the one that will cut down the number of rows being considered the most.
- If the predicates individually don’t cut the results down much, but do so together, that’s a good sign a multi-column index will be useful.
- But first, let’s consider a single column.
Is the column a small scalar?

- int, bigint, float, UUID, datetime(tz)… (but see later for inet and char types).
  -UUIDs have special considerations in B-tree indexes.
- Is the value a primary key or otherwise UNIQUE?
  - If so, B-Tree.
- Is it monotonically increasing on a large, rarely updated table, and the query is doing a range operation?
  - If so, BRIN.
- Otherwise, B-Tree.
  - If the index is primarily to support ORDER BY … DESC, create as descending; otherwise, ascending.
Is the column a text field?

- `varchar()`, `text`, or `char` (if you’re weird).

- Are you doing full-text search, trigrams, or other fuzzy search techniques?
  - Trick question! See later.

- Is the data structured (and prefix-heavy) and you are typically doing prefix searches? (URLs are a typical case here.)
  - Consider SP-GiST.

- Is the value generally small (< 200 characters), or do you require total ordering?
  - If so, B-Tree.

- Otherwise, consider a Hash index.
Is the column a bytea?

- **Why** are you indexing a bytea?

- Don’t do this.

- **Please.**

- If you must, use Hash or calculate a hash and store it separately.
Is the column a range or geometric type?

- GiST is there for you.
- PostGIS indexes are all GiST-based.
- If you need nearest-neighbor searching, GiST for sure.
  - The “Starbucks problem.”
- Experiment with SP-GiST to see if it is a good fit for your data distribution.
Is the column type inet?

• Are you just doing equality?
  • B-Tree
  • (Try Hash to see if it works better for you.)

• Are you doing prefix searches?
  • Consider SP-GiST.
Is the column an array or JSONB?

• Are you just doing equality?
  • Hash.

• Are you searching for key values?
  • GIN.
Is the column JSON-no-B?

• Why is the column JSON?

• Expression index is the only option here.

• If you need indexing, far better to convert it to JSONB.
Are you doing full-text or fuzzy search?

- Full text search: Create a tsvector from the text, and create a GIN index on that.
  - Either store as a separate column, or use an expression index.
  - Separate columns are better for complex tsvector creation.

- Fuzzy search: Create an index on the column using gist_trgm_ops (part of the pg_trgm contrib package).
Is there more than one column in the predicate?

• Consider creating a multi-column index, if the predicates together are highly selective.

• Remember that in an index on (A, B), PostgreSQL will (almost!) never use it for just a search on B.

• Find the right index type for each column individually, and create the index based on the most selective column.

• If one column requires a GiST index, you can use the `btree_gist` package to get GiST operators for basic scalar types.
Is there more than one column in the predicate?

• If the query pattern is an arbitrary equality comparison of the various columns, consider a Bloom index.

• Not uncommon with a GUI-driven search filter.

• If the predicates are selective independently, two indexes might be superior… test!
Does the query contain an expression?

• Consider creating an expression index.

• For example, an index on `unaccent(lower(name))` instead of querying on it.

  • Don’t forget the citext type for the `lower()` problem, though.

• Be sure that particular expression is very heavily queried.

• If you index on a user-written function, make sure it really is IMMUTABLE, not just declared that way.
Is one predicate highly selective?

- SELECT * FROM orders WHERE customer_id = 12 AND active;
  
  - … where only 10% of orders are “active”.

- Consider creating a partial index.
  
  - CREATE INDEX ON orders(customer_id) WHERE active;

- Only contains the rows that match the predicate.

- Can significantly speed up index queries.
Tools.
Do we need an index?

- pg_stat_user_tables.

- Look for tables with a significant number of sequential scans.

- Not all sequential scans are bad! Dig into the particular queries, look at their execute plans.

- pg_stat_statements, the text logs, and pgbadger are your friends here.
Will an index help?

- https://github.com/HypoPG/hypopg

- Allows creation of “hypothetical” indexes.

- Create index, EXPLAIN the query, see if it is being used.

- “Being used” and “makes the query faster” are not always the same thing.

- RDS, at least, supports it.
Is the index being used?

- `pg_stat_user_indexes`.
- Look for indexes that aren’t being used.
- Drop indexes that aren’t benefiting you.
- Indexes have a large intrinsic cost in disk space and UPDATE/INSERT time.
Are indexes bloated?

- Indexes can suffer from bloat.

- VACUUM can’t always reclaim space efficiently, due to index structure.

- Periodic index rebuilds are worth considering.

- [https://github.com/pgexperts/pgx_scripts/blob/master/bloat/index_bloat_check.sql](https://github.com/pgexperts/pgx_scripts/blob/master/bloat/index_bloat_check.sql)
Are indexes corrupted?

• It doesn’t happen often, but it does happen.

• Errors during queries, etc.

• PostgreSQL 10+ has `amcheck`.

• Easy to fix! Drop and recreate the index.
To Conclude
Indexes are great.

- Remember that they are an optimization.
- Always create in response to particular query situations.
- Experiment! Test different index types to see what works best.
- Pick the right index type for the data… don’t just go with B-Tree by default.
- Monitor usage and size to keep the database healthy and trim.
Thank you!
Questions?
Christophe Pettus

CEO, PostgreSQL Experts, Inc.

christophe.pettus@pgexperts.com

twitter @xof

thebuild.com